COMP 302 Winter 2025 Problem Set 2

Problem 1: Factorial with Exception

Implement a function to compute factorial that raises an exception on negative input and explain why this function has type `int -> int`.

Problem 1 Solution

```
1  exception Negative_input
2
3  let rec factorial n =
4    if n < 0 then raise Negative_input
5    else if n = 0 then 1
6    else n * factorial (n - 1)
```

- **Raising an Exception**: The inclusion of an exception for negative input (`raise Negative_input`) does not affect the type signature of the function. In OCaml, the `raise` function can technically be seen as having the type `'a -> 'b` because it can appear in any part of the function without affecting the expected return type. This is because `raise` never actually returns; it interrupts the normal flow of execution.

Problem 2: N-ary Tree Recursive Search

Implement a higher-order function to find a node that satisfies a predicate for an n-ary tree where each node has a list of subtrees.

Problem 2 Solution

```
1  exception NotFound
2
3  type 'a ntree = Empty | Node of 'a * 'a ntree list
4
5  let rec find p t = match t with
6    | Empty -> raise NotFound
7    | Node (x, ts) -> if p x then x else find_in_subtrees p ts
8
9  and find_in_subtrees p ts = match ts with
10   | [] -> raise NotFound
11   | t::ts -> try find p t with NotFound -> find_in_subtrees p ts
```

Problem 3: Recursive Search with Embedded Helper Function

Perform a recursive search on a tree similar to Problem 2 but embed a helper function using `let-in` to manage the recursive logic.

Problem 3 Solution

```
1  exception NotFound
2
3  type 'a ntree = Empty | Node of 'a * 'a ntree list
4
5  let find_with_helper p t =
6    let rec find_internal t = match t with
7      | Empty -> raise NotFound
8      | Node (x, ts) ->
9          if p x then x
10          else find_in_subtrees ts
11    and find_in_subtrees ts = match ts with
12      | [] -> raise NotFound
13      | t::rest ->
14          try find_internal t with
15          | NotFound -> find_in_subtrees rest
16    in find_internal t
```

Problem 4: Recursive Search with Fold

Perform a recursive search on a tree like in Problem 2, but fold list recursion with tree recursion to streamline the recursive process.

Problem 4 Solution

```
1  exception NotFound
2
3  type 'a ntree = Empty | Node of 'a * 'a ntree list
4
5  let rec find_folded p t = match t with
6    | Empty -> raise NotFound
7    | Node (x, ts) when p x -> x
8    | Node (_, []) -> raise NotFound
9    | Node (x, t::ts) ->
10       try find_folded p t
11       with NotFound -> find_folded p (Node (x, ts))
```

Problem 5: Coffee System

You are tasked with implementing a coffee system Each customer can open an account to track their coffee purchases. For every 5th coffee purchase, the coffee should be free. Otherwise, the coffee costs a fixed price.

Requirements

1. Implement a function to open a new coffee account. Each account tracks the number of coffees a customer has purchased.

2. Implement a function to simulate the purchase of coffee. The function should return the price of the coffee (e.g., 2 units per coffee, and free on every 5th purchase).

3. Implement a function that returns the total number of coffees purchased on an account.

Function Signatures

```
1  val open_coffee_account : unit -> coffee_account
2  val buy_coffee : unit -> int
3  val get_purchases : unit -> int
```

Example Usage and Expected Output

```
1  let my_account = open_coffee_account ();;
2
3  my_account.buy_coffee ();;    (* Returns 2 *)
4  my_account.buy_coffee ();;    (* Returns 2 *)
5  my_account.buy_coffee ();;    (* Returns 2 *)
6  my_account.buy_coffee ();;    (* Returns 2 *)
7  my_account.buy_coffee ();;    (* Returns 0, indicating free coffee *)
8  my_account.get_purchases ();;  (* Returns 5 *)
```

Problem 5 Solution

```
1  type counter = {
2    increment : unit -> unit;
3    get_count : unit -> int;
4  }
5
6  let make_counter () =
7    let count = ref 0 in {
8    increment = (fun () -> count := !count + 1);
9    get_count = (fun () -> !count);
10  }
11
12  type coffee_account = {
13    buy_coffee : unit -> int;
14    get_purchases : unit -> int;
15  }
16
17  let create_coffee_account () =
18    let my_counter = make_counter () in
19    let buy_coffee () =
20      my_counter.increment ();
```

```
21      if my_counter.get_count () mod 5 = 0 then 0
22      else 2
23    in
24    let get_purchases () = my_counter.get_count () in
25    { buy_coffee; get_purchases }
```

Problem 6: Simulating Mutable Lists

Simulate mutable lists using an immutable one.

```
1  type 'a mut_list = {
2      append : 'a -> unit;
3      drop : int -> unit;
4      get_list : unit -> 'a list;
5  }
6
7  let make_mut_list l =
8      let v = ref l in
9      (* Implement *)
```

Example Usage and Expected Output

```
1  (* Creating a mutable list with initial elements *)
2  let myList = make_mut_list [1; 2; 3; 4; 5];;
3
4  (* Appending an element to the list *)
5  myList.append 6;;
6  myList.get_list ();;   (* Returns [1; 2; 3; 4; 5; 6] *)
7
8  (* Dropping the first element *)
9  myList.drop 2 ;
10 myList.get_list ();;   (* Returns [3; 4; 5; 6] *)
```

Problem 6 Solution

```
1  type 'a mut_list = {
2      append : 'a -> unit;
3      drop : int -> unit;
4      get_list : unit -> 'a list;
5  }
6
7  let make_mut_list l =
8      let v = ref l in
9      let append x = v := !v @ [x] in
10
11     let drop n =
12       v := let rec drop_aux i lst = match lst with
13         | [] -> []
14         | _ :: tl -> if i > 0 then drop_aux (i - 1) tl else lst
15       in drop_aux n !v
16     in
17     let get_list () = !v in
18     { append; drop; get_list }
```